Channels as Objects in Concurrent Object-Oriented Programming

Joana Campos
Lasige and Department of Informatics
University of Lisbon, Portugal

Vasco T. Vasconcelos
Lasige and Department of Informatics
University of Lisbon, Portugal

There is often a sort of a protocol associated to each class, stating when and how certain methods should be called. Given that this protocol is, if at all, described in the documentation accompanying the class, current mainstream object-oriented languages cannot provide for the verification of client code adherence against the sought class behaviour. We have defined a class-based concurrent object-oriented language that formalises such protocols in the form of *usage types*. Usage types are attached to class definitions, allowing for the specification of (1) the available methods, (2) the tests clients must perform on the result of methods, and (3) the object status — linear or shared — all of which depend on the object's state. Our work extends the recent approach on modular session types by eliminating channel operations, and defining the method call as the single communication primitive in both sequential and concurrent settings. In contrast to previous works, we define a single category for objects, instead of distinct categories for linear and for shared objects, and let linear objects evolve into shared ones. We introduce a standard sync qualifier to prevent thread interference in certain operations on shared objects. We formalise the language syntax, the operational semantics, and a type system that enforces by static typing that methods are called only when available, and by a single client if so specified in the usage type. We illustrate the language via a complete example.

1 Motivation

Today's mainstream object-oriented languages statically check code against object interfaces that describe method signatures only. There are often semantic restrictions in the implementation of classes that impose particular sequences of legal method calls and aliasing restrictions, which clients must observe in order for objects to work properly. Usually, these usage protocols are only described in informal documentation, and hence any disregard cannot be statically detected, revealing itself as a runtime exception (in languages equipped with built-in support to handle exceptional events) or, worse still, in memory corruption and unexpected behavior.

In this paper, we present our work on the language MOOL, a mini object-oriented language in a Java-like style with support for concurrency, that allows programmers to specify usage protocols. The language includes a simple concurrency mechanism for thread spawning. Class protocols are formalised as *usage types* that specify (1) the available methods, (2) the tests clients must perform on the values returned by methods, and (3) the existence of aliasing restrictions, all of which depend on the object's state.

Aliasing makes it difficult for a client to know if it is complying with the protocol since multiple references may exist to the same object, and may alter its state. In our approach, we define a single category for objects, as opposed to distinct categories for linear and for unrestricted (or shared) objects. We let the current status of an object to be governed by its type, allowing linear objects to evolve into shared ones (*cf.* [21, 22]). The opposite is not possible, as we do not keep track of the number of references to a given object. We propose that the programmer explicitly introduces annotations in the

K. Honda and A. Mycroft (Eds.): Programming Language Approaches to Concurrency and communication-cEntric Software 2010 (PLACES'10)
EPTCS 69, 2011, pp. 12–28, doi:10.4204/EPTCS.69.2

usage descriptor, in order to distinguish between **lin** and **un** object states. The **lin** qualifier describes the status of an object that can be referenced in exactly one thread object. The **un** qualifier stands for unrestricted, or shared, and governs the status of an object that can be referenced in multiple threads. The type system further maintains that unrestricted objects do not contain more restrictive linear attributes.

In MOOL, we adopt an approach similar to the one of Gay *et al.* [8] defining a global usage specification of the available methods. We spread the implementation of a usage type over separate methods and classes, following a modular programming principle. This means that we also have to handle *non-uniform objects*, that is, objects that dynamically change the set of available methods. Nierstrasz [17] was the first to study the behaviour of non-uniform, or active, objects in concurrent systems. We extend Gay's work in two ways. (1) The foregoing work treats communication channels shared by different threads as objects, by hiding channel primitive operations in an API from where clients can call methods. In MOOL, we eliminate channels and define a programming language that relies on a simpler communication model – message passing in the form of method calls, both in the sequential and in the concurrent setting. (2) The foregoing work deals with linear types only. In MOOL, we deal with linear types as well as shared ones, treating them in a unified category.

Setting a single category for objects introduces an additional complication in the control of concurrent access to shared objects. Since our main focus are linear objects, we adopt a standard and straightforward solution similar to the *synchronized* methods used in Java in order to enforce serialised access to methods that manipulate shared data, and should not be executed by two threads simultaneously.

Summing up the main contributions of our approach regarding session types for objects:

- In contrast to other works [4, 5, 8], we elect method invocation as the only communication model in both concurrent and sequential programming;
- We annotate classes with a usage descriptor to structure method invocation, and we enhance it with **lin**/**un** qualifiers for aliasing control, thus defining a single category for objects that may evolve from a linear status into an unrestricted one;
- In contrast to other works on session types [8, 13], we replace the shared channel primitive (on which sessions are started) by a conventional **sync** method modifier, enabling multiple threads to independently work on shared operations without interfering with each other.

More examples, detailed explanations of the core language and of its implementation in a prototype compiler, as well as a more complete overview of the literature, are given in the MSc thesis of the first author [2]. The remaining sections are organised as follows: Section 2 introduces the language via an example; Section 3 gives an overview of the language technicalities; and Section 4 contains additional discussion on related work, as well as an outline of future work.

2 The Auction System

Our auction system, adapted from [20, 21], features three kinds of participants: the auctioneer, the sellers and the bidders. Sellers sell items for a minimum price. Bidders place bids in order to buy some item for the best possible price. The auctioneer controls these interactions.

The system is best described by the UML sequence diagrams in Figures 1 and 2, modelling the two main scenarios through a sequence of messages exchanged between objects. The first diagram describes the scenario for a seller, while the second one describes the scenario for a bidder. We are mainly interested in visualising the interaction between the different players; we do not represent concurrent

communication, even though it should not be hard to imagine several seller and bidder threads concurrently making requests to the same auctioneer object. To lighten the representation, we omit the usual dashed open-arrowed line at the end of an activation box indicating the return from a message and its result. Instead, we simply annotate sent messages with the returned value, if any. A conditional message is shown by preceding the message by a conditional clause in square brackets. The message is only sent if the clause evaluates to true.

The first diagram (Figure 1) depicts how a Seller initiates the interaction with the Auctioneer indicating the item to auction and its price. The Auctioneer, after creating an Auction object, where the bids for the item being sold are going to be placed, delegates the service to a new Selling object. Once the auction is set, the auctioneer can start receiving bids for that item. We signal with a note the moment when the two scenarios interweave. In the interaction initiated by a Bidder (Figure 2), the Auctioneer receives a request for the item searched and, if the item is being auctioned, it delegates the service to the Bidding object. We can see that a Bidder holds its own Bidding object from which it can obtain the item initial price as defined by the Seller. Based on the returned value, the Bidder decides whether to make a bid by calling the appropriate method on the Bidding object. Back to the seller scenario, if the sale is successful, the Selling object allows a Seller to obtain the sale final price by invoking method getFinalPrice.

Both scenarios depict a similar pattern, and it is not difficult to conclude from the diagrams that the usage protocols specified in the Selling and Bidding classes should be described by linear types. That is the only way we can guarantee that clients (sellers and bidders) fully obey the specification, by enforcing that their types are consumed to the end. A fresh Selling (and Bidding) object is created at the beginning of each selling (and bidding) interaction, and is implicitly destroyed at the end. We signal object destruction in the diagram to increase the expressiveness of the representation; the type system provides crucial information to object deallocation.

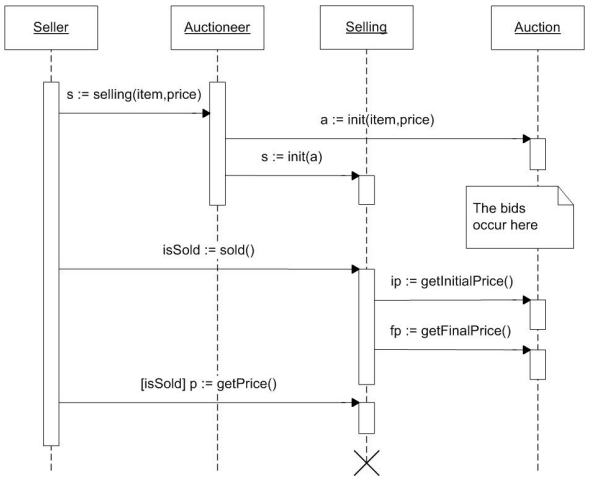

Figure 1: The scenario for a seller

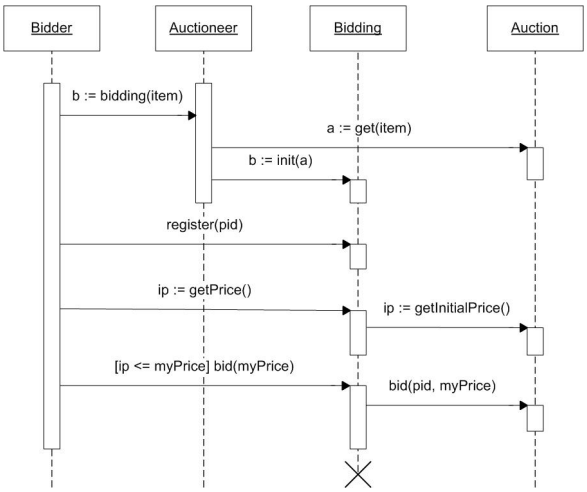

Figure 2: The scenario for a bidder

Usage syntactic details Apart from the usage specification at the beginning of each class, our language presents a typical Java-like syntax. Before explaining the implementation in detail, we introduce some less obvious syntactic details. The usage formalises how clients should use an object of a given class. All methods that are not referred in the usage specification are not visible to clients. For the same reason,

class fields are also private, and cannot be used by other classes.

If a program defines a conventional class named C with methods m1, m2 and m3, and no usage declaration, our compiler will insert **usage** $*${m1 + m2 + m3} as the class default usage type, where each method (m1, m2 and m3) is always available. Formally, this usage defines a recursive branch type of the form μ X.**un**{m1.X + m2.X + m3.X}. The **un** qualifiers are omitted in the examples. A choice between calling one of the three available methods is indicated by (+). Because of the particular form of the recursive type, calling any of these methods on an instance of C will not change the object state nor the set of available methods.

A typical usage declaration for a linear object is a sequential composition of available methods. If C is linear, **usage lin** m1; **lin** m2; **lin** m3; **end**; is a possible usage declaration. Calling methods in the prescribed order on an instance of this class changes the object state and the set of available methods. State **end** is an abbreviation for **un** {}. When an object is in this state, it means that the set of available methods is empty (the usage protocol is finished).

A variant type, denoted by $\langle \ldots + \ldots \rangle$, is indexed by the true and false values returned by the method to which the variant is bound. A client should test the result of the call: if **true** is returned, the new object state, and the available methods, are to be found on the left-hand side of the variant; if **false** is returned, it is the right-hand side to dictate the object state and available methods.

Programming with usage types Consider now the usage specification in Figure 3. When an object of the Auctioneer class is created through the explicit constructor init , only one reference exists to it, but then the object evolves into an unrestricted type, allowing several sellers and bidders to hold references to the same instance. Notice that we have defined a recursive (shared) type. Each client object can do one of two things: (1) it can call method selling to obtain an object that provides an implementation of the selling activity on the auctioneer; or (2) it can call method bidding and obtain an object that implements the buying activity on the auctioneer. Then, it can repeat the interaction all over again: a seller can lower the price of an item with no bids, and start a new sale; a bidder can bid a higher price. The type never ends, and this illustrates why, in any program, we cannot keep track of the number of references to a shared type. The return types of these two methods are Selling [Sold] and Bidding[Register] respectively, because the types of the returned objects have advanced during the execution of each method body. In the example, the state Sold is short for **lin** sold ; ⟨getPrice ; **end** + **end**⟩; (lines 3 and 4 in Figure 6), where (;) binds stronger than (+), and is used for convenience to replace the full type above. In declarations, variable types are shortened (for space reasons), but their full types are implied.

Notice, still in Figure 3, that when a new selling request is made, a new Auction object is created (line 10). This instance is then added to the AuctionMap object (whose class we omit), where the Auctioneer keeps all the auctions (line 12), and is passed to the constructors of both the Selling and Bidding objects. In the selling method, the reference to the Auction is created locally within the method body and assigned to the variable named a (line 10), while in the bidding method, it must be fetched from the AuctionMap object (line 20). It is through reading and writing to this shared Auction object that the protocol takes place.

The usage declaration in the Auction class (Figure 5) shows another example of a recursive type in a shared object. Notice also the **sync** method qualifier (line 10) that is used to control concurrent bids made by separate Bidder threads. This annotation also qualifies the put and get operations in the AuctionMap class (omitted).

Figures 4 and 6 implement two linear types. The usage declaration of the Seller class (Figure 4) is an abbreviation for the nested composition of branch types **lin** { init ; **lin** {run; **un**{}}}. An example of a

```
class Auctioneer {
  usage lin init;
        *{selling + bidding};
  AuctionMap map;
  unit init() {
    map = new AuctionMap();
  }
  Selling[Sold] selling(string item,
                int price) {
    Auction a = new Auction();
    a.init(item, price);
    map.put(item, a);
    Selling s = new Selling();
    s.init(a);
    s; // return s
  }
  Bidding[Register] bidding(
                    string item) {
    Bidding b = new Bidding();
    b.init(map.get(item));
    b; // return b
  }
}
```

Figure 3: An auctioneer

```
class Seller {
  usage lin init; lin run; end;
  string item; int price;
  Auctioneer a;
  unit init(Auctioneer a,
       string item, int price) {
    ... // initialize fields
  }
  unit run() {
    Selling[Sold] s =
          a.selling(item, price);
    // wait for the auction to end
    ...
    if(s.sold())
      print("made " +
        s.getPrice() + " euros!");
    else if(lowerPrice())
      run();
  }
  boolean lowerPrice() {
    ... // implementation omitted
  }
}
```

Figure 4: A seller

```
class Auction {
  usage lin init; Choose
   where Choose = *{bid + getInitialPrice
           + getFinalPrice + getBidder};
  string item; int initPrice;
  int bidder; int finalPrice;
  unit init(string item, int initPrice) {
    ... // initialize fields
  }
  sync unit bid(int pid, int bid) {
    if(finalPrice <= bid) {
      bidder = pid;
      finalPrice = bid;
    }
  }
  ... // the getters
}
```

Figure 5: An auction

```
class Selling {
  usage lin init; Sold
    where Sold = lin sold;
                 ⟨getPrice; end + end⟩;
  Auction[Choose] a; int finalPrice;
  unit init(Auction[Choose] a) {
    ... // initialize fields
  }
  boolean sold() {
    finalPrice = a.getFinalPrice();
    finalPrice >= a.getInitialPrice();
  }
  int getPrice() { finalPrice; }
}
```

Figure 6: A selling protocol

```
class Main {
  unit main() {
    Auctioneer a = new Auctioneer();
    a.init();
    Seller seller = new Seller();
    seller.init(a, "psp", 100);
    spawn seller.run();
    Bidder bidder1 = new Bidder();
    bidder1.init(a, 1, 100);
    spawn bidder1.run();
    Bidder bidder2 = ...
  }
}
```

Figure 7: The main class

(class declarations)	$D ::= \mathsf{class}\ C\ \{u; \vec{F}; \vec{M}\}$
(field declarations)	$F ::= t\ f$
(method declarations)	$M ::= s\ t\ m(t\ x)\ \{e\}$
(method qualifiers)	$s ::= \varepsilon \mid \mathsf{sync}$
(types)	$t ::= \mathsf{unit} \mid \mathsf{boolean} \mid C[u]$
(values)	$v ::= \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid o$
(expressions)	$e ::= v \mid x \mid o.f$
	$\mid e;e \mid o.f = e$
	$\mid \mathsf{new}\ C() \mid o.m(e) \mid o.f.m(e)$
	$\mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \mid \mathsf{while}\ (e)\ e$
	$\mid \mathsf{spawn}\ e$
(class usage types)	$u ::= q\ \{m_i; u_i\}_{i \in I} \mid \langle u + u \rangle \mid$
	$\mid X \mid \mu X.u$
(type qualifiers)	$q ::= \mathsf{lin} \mid \mathsf{un}$

Figure 8: User syntax

variant type is provided in the Selling class (line 4 in Figure 6). A variant type is always linear, so the redundant **lin** qualifier can be omitted. This type requires that the client evaluates the returned boolean value of the sold method in order to determine the next available method: if the value evaluates to **true**, then the caller can obtain the price via getPrice (because the item was sold), otherwise the interaction ends.

The lowerPrice method in the Seller class (Figure 4) is not referred in the usage specification (and is thus omitted from the object type). Our type system ensures that only methods in the usage type are visible to clients (*cf.* [8]). All methods not specified in the usage type can be accessed from their own classes, but never alter the type (otherwise client views of the object could become inconsistent).

Finally, the Main class in Figure 7 creates an Auctioneer object that controls the auction, and spawns a separate thread for a Seller and two Bidder objects, using a Java-like technique for thread creation.

3 The Core Language

Most of the technicalities that we now present are based on the core language from modular session types by Gay *et al.* [8], and on the linear type system presented by Vasconcelos [22]. Inspired by these approaches to sessions types, our main challenge was the attempt to formalise the convergence of channels and objects in a simple concurrent object-oriented language.

Syntax In MOOL, we distinguish between the user syntax and the runtime syntax. The former is the programmer's language, and is defined in Figure 8; the latter is only required by the operational

$$
\begin{array}{rrl}
\text{(values)} & v & ::= \ \ldots \mid \mathsf{uninit}_{\mathsf{C[u]}} \\
\text{(expressions)} & e & ::= \ \ldots \mid \mathsf{insync}\ o\ e \\
\text{(contexts)} & \mathcal{E} & ::= \ [\_] \mid \mathcal{E};e \mid o.f = \mathcal{E} \\
 & & \quad \mid o.m(\mathcal{E}) \mid o.f.m(\mathcal{E}) \\
 & & \quad \mid \mathsf{if}\ (\mathcal{E})\ e\ \mathsf{else}\ e \mid \mathsf{while}\ (\mathcal{E})\ e \\
\text{(lock flags)} & l & ::= \ 0 \mid 1 \\
\text{(object records)} & R & ::= \ (t, l, \vec{f} = \vec{v}) \\
\text{(heaps)} & h & ::= \ \emptyset \mid h, o = R \\
\text{(states)} & S & ::= \ (h;\ e_1 | \ldots | e_n)
\end{array}
$$

Figure 9: Runtime syntax

semantics, and appears in Figure 9.

In order to simplify both the operational semantics and the type system, we introduce some restrictions on the syntax of specific constructs: (1) an assignment expression is only defined on fields (not on parameters); (2) a method call is only defined on a field of an object reference (not on an arbitrary expression); (3) it follows from the above that calling a method on a parameter requires assigning it first to a field; and (4) methods accept exactly one parameter.

We assume that class identifiers in a sequence of declarations $\vec{D}$ are all distinct, and that the set of method and field names declared in each class contains only distinct names as well. Object references o include the keyword this, which references the current instance. In the examples, field accesses omit the prefix this; the compiler can insert it when needed. Method modifiers, which include the sync keyword, are optional, which we indicate in the syntax by the empty string ε.

The MOOL syntax defines non-object and object types. Non-object types include the primitive unit and boolean types. The unit type has the single value unit, while the boolean type comprises two values: true and false. The type of an object is $C[u]$ in the style of modular session types [8], describing an object of a class named C at usage type u. From a client class perspective, it prescribes the structure of method invocation: in which order/how methods should be called.

The MOOL expressions are standard in object-oriented languages. We have defined (in order of appearance) values, parameters, fields, the sequential expression composition, assignment to fields, object creation, the self-call, the method call on fields, control flow expressions, and thread creation using the spawn primitive. The form of the spawn expression means that the inner expression e is evaluated in a newly created thread.

In the core language, only the abstract syntax of usage types is defined, omitting the **usage** and **where** clauses, and the branch choice (+) and recursive (*) operators that have been introduced in the examples. However, it is not difficult to translate from the syntax of the examples into the core language syntax. For example, the recursive type **usage lin** init ; $*\{$ selling + bidding$\}$; in the Auctioneer class takes the following configuration in the core language: $\mathsf{lin}\{\mathsf{init}; \mu X.\mathsf{un}\{\mathsf{selling}.X, \mathsf{bidding}.X\}\}$.

Branch types are denoted by $\{m_i; u_i\}_{i \in I}$, defining available methods m_i and their continuations u_i. Variant types are denoted by $\langle u + u \rangle$, and are indexed by the two boolean values, true first. In order to resolve the type, the caller must perform a test on the result of the preceding method. For simplicity,

$$\text{(R-CONTEXT)}\ \frac{(h;\ e_1|\ldots|e|\ldots|e_n) \longrightarrow (h';\ e_1|\ldots|e'|\ldots|e_n)}{(h;\ e_1|\ldots|\mathcal{E}[e]|\ldots|e_n) \longrightarrow (h';\ e_1|\ldots|\mathcal{E}[e']|\ldots|e_n)}$$

$$\text{(R-SPAWN)}\ \ (h;\ e_1|\ldots|\mathcal{E}[\mathsf{spawn}\ e]|\ldots|e_n) \longrightarrow (h;\ e_1|\ldots|\mathcal{E}[\mathsf{unit}]|e|\ldots|e_n)$$

Figure 10: Reduction rules for states

we use binary-only variants as in typestates [18], but more generous variants using enumerations can be found in the literature [8]. Recursive types are indicated in the syntax by $\mu X.u$, and need to be *contractive*, which means that no subexpression of the form $\mu X_1.\cdots\mu X_n.X_1$ is allowed [8]. These types are treated in an equi-recursive discipline, which means that we regard $\mu X.u$ and its unrolling $u\{^{\mu X.u}/_X\}$ as equal types.

A usage branch type is annotated with a qualifier q for aliasing control. Recall that we define a single category for objects, as opposed to distinct categories for linear and for unrestricted objects, allowing linear objects to evolve into a status where they can be shared by multiple clients, but not the opposite (*cf.* [21, 22]).

The type system, that we introduce later, imposes some further restrictions on usage types, namely that: (1) the initial class usage type must be a branch, because a variant type is bound to the result of a method call; (2) in a variant type configuration $\langle u+u\rangle$, each u corresponds to a lin-qualified variant (and that is why the qualifier is always omitted); (3) an object final status is always unrestricted, whether because it evolves into an end state (with an empty set of alternatives), or because, being recursive, it never ends, as illustrated in the examples; and (4) unrestricted objects may not contain linear attributes.

The runtime syntax in Figure 9 introduces additional elements required by the operational semantics, but not available in the user syntax. Values include $\mathsf{uninit}_{\mathsf{C[u]}}$, which is used by our compiler to initialise fields of type $C[u]$ when an object is created. Expressions are extended with the insync expression in the style of Flanagan and Abadi [7], denoting that the subexpression e is currently being evaluated while the lock is held on object o.

Evaluation contexts are denoted by $\mathcal{E}$, and are expressions with one hole. Contexts specify the order in which expressions are evaluated, defining where reduction rules can be applied. A heap is viewed as a map (a partial function of finite domain) from object identifiers o into records R. The operation $h, o = R$ adds an entry to the heap h, if o does not yet exist, and is considered to be associative and commutative, which means that a heap is an unordered set of bindings [8]. Object records are instances of usage-typed classes and are represented by the triple $(t, l, \vec{f} = \vec{v})$, where t is the object current type, l represents the object lock flag, and $\vec{f} \mapsto \vec{v}$ is a mapping from field identifiers to their values. The *lock* can be either in a locked or unlocked state, denoted by the 1 and 0 flag, respectively. The record $(\_, 1, \_)$ means that some thread currently holds the lock associated with the current instance. Initially, an object is created with the flag set to 0. States consist of two components: a heap and a parallel composition of expressions sharing the heap.

Operational Semantics Figure 10 defines the reduction rules for states, taking the form of a parallel composition of expressions:

$$(h;\ e_1|\ldots|e_n) \longrightarrow (h';\ e'_1|\ldots|e'_m)$$

Predicates $\mathsf{lin}(v)$ **and** $\mathsf{un}(v)$ **with** $q ::= \mathsf{lin} \mid \mathsf{un}$

- if $v = \mathsf{unit}, \mathsf{true}, \mathsf{false}$ then $\mathsf{un}(v)$
- if $v = o$ and $h(o).\mathsf{usage} = \langle \_ + \_ \rangle$ then $\mathsf{lin}(v)$
- if $v = o$ and $h(o).\mathsf{usage} = q\ \{\ldots\}$ then $q(v)$

Predicates $\mathsf{lin}(t)$ **and** $\mathsf{un}(t)$ **with** $q ::= \mathsf{lin} \mid \mathsf{un}$

- if $t = \mathsf{unit}, \mathsf{boolean}$ then $\mathsf{un}(t)$
- if $t = C[\langle \_ + \_ \rangle]$ then $\mathsf{lin}(t)$
- if $t = C[q\ \{\ldots\}]$ then $q(t)$

Function $\mathsf{init}(t)$

- $\mathsf{init}(\mathsf{boolean}) = \mathsf{false}$
- $\mathsf{init}(\mathsf{unit}) = \mathsf{unit}$
- $\mathsf{init}(C[u]) = \mathsf{uninit}_{\mathsf{C[u]}}$

Figure 11: Auxiliary functions for values and types

R-CONTEXT is standard, defining which expression can be evaluated next in the program execution. It is the rule that invokes all the other reduction rules for expressions (see Figure 12). R-SPAWN detaches the expression to a new thread running in parallel. After spawn, the original thread proceeds to evaluate its next instruction. The value of the spawn expression is unit, as the result of evaluating e is not transferred back to the original thread.

The rules for expressions in Figure 12 take a similar form, with the state containing only one expression:

$$(h;\ e) \longrightarrow (h';\ e')$$

The rules use in their definitions the predicates defined in Figure 11. Predicate $q(v)$ determines the type status (linear or unrestricted) given a value v. If v is a value of a primitive type, then its status is always unrestricted. If v is an object, then its status depends on its current type, which can be fetch from the usage component of the record associated with o in heap h. A variant type is always linear, and the status of a branch type is defined by the programmer. Predicate $q(t)$ follows a similar pattern, with the difference that it takes a type as its argument. Finally, function $\mathsf{init}(t)$ takes a type as its argument and returns the type default value.

R-LINFIELD and R-UNFIELD extract the value of the field from an object in the heap and determine whether the value belongs to an unrestricted or a linear type. After access, the value of an unrestricted field is v as in Java, but the value of a linear field must be unit to ensure that only one reference to an object with a linear type exists at any given moment.

R-SEQ reduces the result to the second part of the sequence of expressions, discarding the first part only after it has become a value. R-ASSIGN updates the value of the field with value v. The value of the entire expression needs to be unit (as opposed to v), and this is again a linearity constraint. R-NEW generates a fresh object identifier and obtains the field names and types from the set of the class fields.

$$\text{(R-LinField)}\ \frac{h(o).f = v \qquad \mathsf{lin}(v)}{(h;\ o.f) \longrightarrow (h\{o.f \mapsto \mathsf{unit}\};\ v)} \qquad \text{(R-UnField)}\ \frac{h(o).f = v \qquad \mathsf{un}(v)}{(h;\ o.f) \longrightarrow (h;\ v)}$$

$$\text{(R-Seq)}\ \ (h;\ v;e) \longrightarrow (h;\ e) \qquad \text{(R-Assign)}\ \ (h;\ o.f\ =\ v) \longrightarrow (h\{o.f \mapsto v\};\ \mathsf{unit})$$

$$\text{(R-New)}\ \frac{o\ \text{fresh} \qquad C.\mathsf{fields} = \vec{t}\ \vec{f}}{(h;\ \mathsf{new}\ C()) \longrightarrow (h, o = (C, C.\mathsf{usage}, 0)\vec{f} = \mathsf{init}(\vec{t});\ o)}$$

$$\text{(R-SelfCall)}\ \frac{\_\ m(\_x)\ \{e\} \in (h(o).\mathsf{class}).\mathsf{methods}}{(h;\ o.m(v)) \longrightarrow (h;\ e\{^{o}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\text{(R-Call)}\ \frac{m \in h(o).f.\mathsf{usage} \qquad \_\ m(\_x)\ \{e\} \in (h(o).f.\mathsf{class}).\mathsf{methods}}{(h;\ o.f.m(v)) \longrightarrow (h;\ e\{^{o.f}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\text{(R-SCall)}\ \frac{\begin{array}{c} h(o).f.\mathsf{lock} = 0 \qquad m \in h(o).f.\mathsf{usage} \\ \mathsf{sync}\ \_\ m(\_x)\ \{e\} \in (h(o).f.\mathsf{class}).\mathsf{methods} \end{array}}{(h;\ o.f.m(v)) \longrightarrow (h\{(o).f.\mathsf{lock} \mapsto 1\};\ \mathsf{insync}\ o\ e\{^{o.f}/_{\mathsf{this}}\}\{^{v}/_{x}\})}$$

$$\text{(R-InSync)}\ \ (h;\ \mathsf{insync}\ o\ v) \longrightarrow (h\{(o).\mathsf{lock} \mapsto 0\};\ v)$$

$$\text{(R-IfTrue)}\ \ (h;\ \mathsf{if}\ (\mathsf{true})\ e'\ \mathsf{else}\ e'') \longrightarrow (h;\ e') \quad \text{(R-IfFalse)}\ \ (h;\ \mathsf{if}\ (\mathsf{false})\ e'\ \mathsf{else}\ e'') \longrightarrow (h;\ e'')$$

$$\text{(R-While)}\ \ (h;\ \mathsf{while}\ (e)\ e') \longrightarrow (h;\ \mathsf{if}\ (e)\ \{e';\mathsf{while}\ (e)\ e'\}\ \mathsf{else}\ \mathsf{unit})$$

Figure 12: Reduction rules for expressions

Notice that a new object record, indexed by the object identifier o, is added to the heap, that the object starts in an unlocked state 0, and that function $\mathsf{init}(t)$ takes a type as its argument and returns its default value.

R-SelfCall is for method calls directly on an object reference, which typically result from calls on this. These calls do not depend on the usage type, unlike rule R-Call. In both rules, if the hypothesis holds, then the method body is prepared by replacing occurrences of this by the receiver heap address (so that occurrences of this within the method body refer to the receiver instance) and the actual parameter by the formal one. The resulting expression is the method body with the above substitutions that can be further reduced using the appropriate rules.

R-SCall is for synchronized calls. The operational semantics proposed by Flanagan and Abadi [7] inspired this and R-InSync. The new element here is the lock acquired on the receiver object, which is released in R-InSync, by updating the lock component with the flag 0. R-IfTrue, R-IfFalse and R-While are standard.

Subtyping The subtyping definition for the MOOL language is similar to the one described for the language in [8]. The subtype usage relation $<:$, defined by co-induction, includes the following rules:

- If $q\{m_i : u_i\}_{i\in I} <: u'$ then $u' = q\{m_j : u'_j\}_{j\in J}$ with $J \subseteq I$ and $\forall j \in J, u_j <: u'_j$
- If $u <: \langle u' + u'' \rangle$ then $u' <: u$ or $u'' <: u$

The branch type is the source of subtyping by allowing that the possible usage choices can safely be used in the context where the superusage is expected. The proof that the usage relation is a pre-order (i.e. a relation that is reflexive and transitive) is not provided here, but can be adapted from [9].

The subtyping usage rule relation is extended to the types of our language as the smallest reflexive relation which includes the following: $C[u] <: C[u']$ if $u <: u'$. This means that implicitly the subtyping relation for the two other types in our language is defined as $\mathsf{boolean} <: \mathsf{boolean}$ and $\mathsf{unit} <: \mathsf{unit}$.

Type System The inference rules that define the type system use two typing environments, Θ and Γ. We consider typing environments as maps (or partial functions of finite domain) similarly to heaps. The typing environment Θ is a map from usage types u to typing environments Γ, and records the field typing environment associated with usage type u. It is used to keep track of visited usage types, thus preventing cycles in the presence of recursive types.

The typing assumptions for the environment Γ have the form:

$$\Gamma \ ::= \ \Sigma \mid \langle \Gamma + \Gamma \rangle$$

where Σ is a map from fields $o.f$ and parameters x to types t. The environment $\langle \Gamma + \Gamma \rangle$ represents a pair of maps: the map on the left is used for the true variant, while the map on the right is for the false one.

Below are the typing judgements where these environments are used. The typing judgement for usage types is presented on the left, and the one on the right is used for expressions:

$$\Theta; \Gamma \rhd_C u \lhd \Gamma' \qquad\qquad \Gamma \rhd e : t \lhd \Gamma'$$

The typing judgement for usage types reads: "A usage type u is valid for a class named C starting from field and parameter types in Γ, the initial environment, and ending with field and parameter types in Γ', the final environment". The initial and final environments may be different, because object field types in Γ may have changed after each method described in u has been type-checked.

The typing judgement for expressions follows a similar pattern. Γ is the initial typing environment before type-checking the expression e, and Γ' is the final one, after associating e with the type t. Again, the initial and final environments may be different, because of side-effects the expression being checked may cause on the environment Γ, turning it into the environment Γ'. In particular, identifiers may become unavailable in the case of references to linear objects, and object types may change as a result of the evaluation of some language constructs, namely method calls and control flow expressions. Using a similar notation to the one used to update the heap, if $\Gamma(o.f)$ is defined, i.e., if $o.f \in dom(\Gamma)$, then $\Gamma\{o.f \mapsto t\}$ denotes the environment obtained by updating to t the type of $o.f$.

The typing rules defined for MOOL have a clear algorithmic configuration, and type-checking is modular and performed following a top-down strategy: a program checking is conducted by checking each class separately, which in turn conducts the checking of each method within the class *in the order in which it appears in the specified usage type*. In what follows, rules are presented in the order in which they are evaluated by the type-checker (and in the order in which they should be read).

Figure 13 defines rules for typing programs. When checking that a program is well-formed, T-PROGRAM simply checks that each class defined in it is well-formed. T-CLASS constructs the field

$$\text{(T-PROGRAM)}\ \frac{\vdash D_1 \quad \ldots \quad \vdash D_n}{\vdash D_1 \ldots D_n} \qquad \text{(T-CLASS)}\ \frac{\emptyset;\mathsf{this}.\vec{f} : \vec{t} \triangleright_C u \triangleleft \Sigma \qquad \mathsf{un}(\Sigma)}{\vdash \mathsf{class}\ C\ \{u;\vec{t}\ \vec{f};\_\}}$$

Figure 13: Typing rules for programs

$$\text{(T-BRANCH)}\ \frac{\begin{array}{c}\forall i \in I \qquad s_i\ t_i\ m_i(t'_i\ x_i)\ \{e_i\} \in C.\mathsf{methods} \qquad q(\Sigma) \qquad \Sigma, x_i : t'_i \triangleright e_i : t_i \triangleleft \Gamma \\ x_i : t''_i \in \Gamma \Rightarrow \mathsf{un}(t''_i) \qquad \Gamma = \langle \_ + \_ \rangle \Rightarrow t_i = \mathsf{boolean} \qquad \Theta;\Gamma \backslash x_i \triangleright_C u_i \triangleleft \Gamma'\end{array}}{\Theta;\Sigma \triangleright_C q\ \{m_i;u_i\} \triangleleft \Gamma'}$$

$$\text{(T-VARIANT)}\ \frac{\Theta;\Gamma' \triangleright_C u_t \triangleleft \Gamma \qquad \Theta;\Gamma'' \triangleright_C u_f \triangleleft \Gamma}{\Theta;\langle \Gamma' + \Gamma'' \rangle \triangleright_C \langle u_t + u_f \rangle \triangleleft \Gamma}$$

$$\text{(T-USAGEVAR)}\ (\Theta, X : \Gamma);\Gamma \triangleright_C X \triangleleft \Gamma' \qquad \text{(T-REC)}\ \frac{(\Theta, X : \Gamma);\Gamma \triangleright_C u \triangleleft \Gamma'}{\Theta;\Gamma \triangleright_C \mu X.u \triangleleft \Gamma'}$$

Figure 14: Typing rules for classes

arguments in the initial typing environment. Notice that when starting to type-check a class, the environment Θ is empty as no usage types have yet been visited. T-CLASS is responsible for calling the rules for type-checking usage type constructs (see Figure 14). After type-checking u, the object fields in Σ are a subset of the original ones, that have been consumed, inside the object, to the end. Those which are missing have been passed as parameters, or returned from methods, and thus removed from the environment. This means that the object fields in a class final environment have always unrestricted types, either an unrestricted recursive type or un end. Function $\mathsf{un}(\Sigma)$ recursively calls predicate $q(t)$, passing each field type in Σ as argument, in order to guarantee that no linear fields exist in the class final environment. We also define $q(\Sigma)$ if and only if $o.f : t \in \Sigma$ implies $q(t)$, so that the type system can check that no linear fields are allowed in shared objects.

The rules in Figure 14 describe how the four usage constructs (*cf.* Figure 8) are typed in MOOL. T-BRANCH conducts the type-checking of the methods defined in the class usage type in the order in which they appear in the specified sequence. The appropriate rules for expressions (see Figures 15 and 16) are called by T-BRANCH so that each method body e_i can be type-checked. The initial environment is a single map of fields and parameters, denoted by Σ, while the final one may be a pair of maps if the method m_i returns a value of type boolean. Thus Γ is used instead. The relation $q \subseteq q'$ is reflexive and transitive, and in particular $\mathsf{lin} \subseteq \mathsf{un}$. When exiting the method body, the parameter type in Γ may have changed from the initial type t'_i to the final one t''_i. Additionally, the rule requires that t''_i is unrestricted, so the function $\mathsf{un}(t''_i)$ checks that the parameter has been consumed to the end. Finally, the parameter entry must be removed from Γ before the rule advances to type-checking the method continuations u_i, where again the field typing may change, modifying the initial environment Γ to the final one Γ'.

In checking variant types, T-VARIANT requires a consistency between the variants final environment Γ, after passing Γ', the left environment, to type the true variant, and Γ'', the right environment, to type the false variant. T-USAGEVAR simply reads the type variable X from map Θ. Because X represents an

$$\text{(T-LinVar)}\ \frac{\mathsf{lin}(t)}{\Sigma, x : t \triangleright x : t \triangleleft \Sigma} \qquad \text{(T-UnVar)}\ \frac{\mathsf{un}(t)}{\Sigma, x : t \triangleright x : t \triangleleft \Sigma, x : t}$$

$$\text{(T-LinField)}\ \frac{\mathsf{lin}(t)}{\Sigma, o.f : t \triangleright o.f : t \triangleleft \Sigma} \qquad \text{(T-UnField)}\ \frac{\mathsf{un}(t)}{\Sigma, o.f : t \triangleright o.f : t \triangleleft \Sigma, o.f : t}$$

$$\text{(T-Seq)}\ \frac{\Gamma \triangleright e : t \triangleleft \Gamma'' \qquad \Gamma'' \triangleright e' : t' \triangleleft \Gamma'}{\Gamma \triangleright e; e' : t' \triangleleft \Gamma'} \qquad \text{(T-Assign)}\ \frac{\Sigma \triangleright e : t \triangleleft \Sigma' \qquad \Sigma'(o.f) = t \qquad \mathsf{un}(t)}{\Sigma \triangleright o.f \ = \ e : \mathsf{unit} \triangleleft \Sigma'}$$

$$\text{(T-New)}\ \ \Gamma \triangleright \mathsf{new}\ C() : C[C.\mathsf{usage}] \triangleleft \Gamma \qquad \text{(T-Spawn)}\ \frac{\Gamma \triangleright e : t \triangleleft \Gamma' \qquad \mathsf{un}(t)}{\Gamma \triangleright \mathsf{spawn}\ \ e : \mathsf{unit} \triangleleft \Gamma'}$$

Figure 15: Typing rules for simple expressions

infinite branch in the usage type tree that no program can ever consume to the end, the rule cannot define the final environment as being the same as the initial one, which means that the final environment can be whichever we want. T-Rec, not only reads the type from Θ, but also checks the type against the class, using T-UsageVar to type usage variables X, and the other two rules, T-Branch and T-Variant, to type branch and variant constructs. The final environment Γ' may be different, because field types may have changed, after type-checking expressions in method bodies (recall that T-Branch calls the appropriate rules for expressions).

We now present the rules for expressions. In order to make them more readable, we use the single environment Σ everywhere a map is needed to index a field or a parameter; otherwise the more general Γ is used. Figures 15 and 16 define the rules for the remaining expressions of the top level language.

T-LinVar and T-UnVar evaluate the types of method parameters, distinguishing linear from unrestricted ones. Recall that, as we cannot call methods on parameters, they must be first assigned to fields. In the case of parameters with linear types, the type system requires that they are removed from the environment, so that they can no longer be used after assignment. Predicates $\mathsf{lin}(t)$ and $\mathsf{un}(t)$ are used to determine the status of a given type. In practice, primitive types (unit and boolean) are always unrestricted; for object types the current status varies with the type. T-LinField, T-UnField are similar to the rules defined above for parameters, removing fields with linear types from the environment.

Typing the sequential composition of expressions is simple: T-Seq considers the typing of the second subexpression taking into account the effects of the first one on the environment, which may be different when e represents a method call or a control flow expression. T-Assign formalises assignments to fields. The type of the reference and the expression must be consistent and, because of linearity, it must be unrestricted. The type of the entire assignment expression is unit (as opposed to t) to enforce that a linear reference goes out of scope after appearing on the right-hand side of the assignment. T-New simply states that a new object has the initial usage type declared by its class. T-Spawn requires not only that the thread body is typable, but also that it is not of a linear type; otherwise one could create a linear reference and not use it to the end (suppose we wrote **spawn new** C()). The type of the entire spawn expression is unit, because the evaluation is performed only for its effect (creating a new thread); no result is ever returned.

$$\text{(T-CALL)}\ \frac{\Gamma \rhd e : t \lhd \Sigma \qquad \Sigma(o.f) = C[\_\ \{m_i;u_i\}_{i\in I}] \qquad j \in I \qquad t\ m_j(t\,x)\ \{\_\} \in C.\mathsf{methods}}{\Gamma \rhd o.f.m_j(e) : t \lhd \Sigma\{o.f \mapsto C[u_j]\}}$$

$$\text{(T-IFV)}\ \frac{\Gamma \rhd o.f.m(e) : \mathsf{boolean} \lhd \Sigma \qquad \Sigma(o.f) = C[\langle u_t + u_f \rangle] \qquad \Sigma\{o.f \mapsto C[u_t]\} \rhd e' : t \lhd \Gamma' \qquad \Sigma\{o.f \mapsto C[u_f]\} \rhd e'' : t \lhd \Gamma'}{\Gamma \rhd \mathsf{if}\ (o.f.m(e))\ e'\ \mathsf{else}\ e'' : t \lhd \Gamma'}$$

$$\text{(T-WHILEV)}\ \frac{\Gamma \rhd o.f.m(e) : \mathsf{boolean} \lhd \Sigma \qquad \Sigma(o.f) = C[\langle u_t + u_f \rangle] \qquad \Sigma\{o.f \mapsto C[u_t]\} \rhd e' : t \lhd \Gamma}{\Gamma \rhd \mathsf{while}\ (o.f.m(e))\ e' : \mathsf{unit} \lhd \Sigma\{o.f \mapsto C[u_f]\}}$$

$$\text{(T-IF)}\ \frac{\Gamma \rhd e : \mathsf{boolean} \lhd \Gamma' \qquad \Gamma' \rhd e' : t \lhd \Gamma'' \qquad \Gamma' \rhd e'' : t \lhd \Gamma''}{\Gamma \rhd \mathsf{if}\ (e)\ e'\ \mathsf{else}\ e'' : t \lhd \Gamma''}$$

$$\text{(T-WHILE)}\ \frac{\Gamma \rhd e : \mathsf{boolean} \lhd \Gamma' \qquad \Gamma' \rhd e' : t \lhd \Gamma}{\Gamma \rhd \mathsf{while}\ (e)\ e' : \mathsf{unit} \lhd \Gamma'}$$

$$\text{(T-INJL)}\ \frac{\Gamma \rhd e : t \lhd \Gamma'}{\Gamma \rhd e : t \lhd \langle \Gamma' + \Gamma'' \rangle} \qquad \text{(T-INJR)}\ \frac{\Gamma \rhd e : t \lhd \Gamma''}{\Gamma \rhd e : t \lhd \langle \Gamma' + \Gamma'' \rangle}$$

$$\text{(T-SUB)}\ \frac{\Gamma \rhd e : C[u] \lhd \Gamma' \qquad C[u] <: C[u']}{\Gamma \rhd e : C[u'] \lhd \Gamma'} \qquad \text{(T-SUBENV)}\ \frac{\Gamma \rhd e : t \lhd \Sigma \qquad \Sigma <: \Sigma'}{\Gamma \rhd e : t \lhd \Sigma'}$$

Figure 16: Typing rules for calls and control flow expressions

T-CALL requires that the receiver has an appropriate usage type. The call results in the receiver changing its type to $C[u_j]$, with the final environment Σ being updated accordingly, while at the same time the type advances from $\{m_i;u_i\}$ to u_j.

As opposed to the rules presented thus far, the remaining ones in Figure 16 are not syntax-directed, which means that to implement them additional information is needed as they are evaluated. To simplify the rules, for an object to have a variant type, the method call on which the variant type depends must be performed on the condition of a control flow expression (and not on an arbitrary assignment). Recall that a variant type is tied to the result of a method and, depending on the value returned, an object type may be modified differently. The type system requires that the value is tested on the condition of the expression, so as to have the type immediately deduced. We believe that this restriction does not impair current object-oriented programming practices, as it reflects how a variant type is typically given to an object.

T-IFV and T-WHILEV are particular cases of the more general rules for control flow expressions that we describe below. The order in which we present them here corresponds to the actual order in which the type-checker evaluates them. Both rules depend on T-CALL to deduce the return type of the method

tested on the condition. In the case of T-IFV, both branches use Σ, the environment that results from the call, as their initial environment. The method returned value defines how $o.f$ type is updated, thus determining which branch (the then or the else) shall be executed. However, because only one of the branches can be executed, the rule enforces that the two branches should be consistent, sharing the same final environment Γ'. T-WHILEV follows a similar pattern: a while expression can be reduced to an if that is repeated while a particular condition returns true. The type of the entire loop expression is unit, because its result can never be used. T-IF, T-WHILE are standard rules for control flow expressions, and should be used only after the above rules have been tried.

T-INJL and T-INJR build a variant environment as follows: an environment Γ becomes $\langle \Gamma + \Gamma' \rangle$ by injecting Γ on the left using rule T-INJL, and becomes $\langle \Gamma' + \Gamma \rangle$ by injecting on the right using rule T-INJR. Typically, these rules build the final environment of a boolean method to which a variant is tied.

T-SUB and T-SUBENV are similar to the subsumption rules described for the language in modular session types [8]. T-SUB is a standard subsumption rule that simply says that whenever we can prove that type t' is a subtype of t, we can treat t' as if it were type t. T-SUBENV allows subsumption in the final environment, and is used for the branches of control flow expressions.

4 Concluding Remarks

Session types, introduced in [12, 13, 19], have been proposed to enhance the verification of programs at compile-time by specifying the sequences and types of messages in communication protocols. Traditionally associated with communication channels, session types provide a means to enforce that channel implementations obey the requirements stated by their types. Originally developed for dyadic sessions, the concept was then extended to multi-party sessions [11]. Programming languages that implement session types come in all flavours: pi calculus, an idealised concurrent programming language in the context of which the original concepts were developed, functional languages [16, 21, 23], CORBA [20], object-oriented languages [4, 5, 8, 15]. Channels as conceived in session type theory are special entities that carry messages of different types, bi-directionally, in a specific sequence between two end points. These channels are usually implemented in a socket-like style, and, to our knowledge, none of the previous attempts to integrate session types into object-oriented programming ever abstracted this notion of communication channels. The work on Moose [4, 5], a multi-threaded object-oriented calculus with session types, was the first attempt to marriage the (concurrent) object-oriented paradigm and session types. Unlike our work, this and subsequent work have kept distinct mechanisms for local and remote communication, in the form of method call and channel operations, respectively.

In the literature, several lines of research can be found that reveal similarities with session type theory. One of these lines introduces the concept of typestate [18] in which the state of the object in some particular context determines the set of available operations in that context, based on pre- and post-conditions. Objects, by nature, can be in different states throughout their life cycle. The concept involves static analysis of programs at compile-time so that all the possible states of an object and associated legal operations can be tracked at each point in the program text. Typestate checking has been incorporated in several programming languages [1, 3, 6], and some ideas relate very closely to session type recent approach on modularity.

This paper formalises a mini class-based language that uses primitives consistent with most object-oriented languages, and incorporates support for the specification and static checking of usage protocols. The existing work on modular session types [8] has been the inspiration for our specification language and type system, where we replace operations on communication channels by remote method invoca-

tions. Another distinguishing feature comes from allowing objects to change from a linear status to a shared one. We also use a standard synchronization mechanism to control concurrency, instead of the well-known shared channel on which linear channels are created. We have designed a simple operational semantics and a static type system which checks client code conformance to method call sequences and behaviourally constrained aliasing, as expressed by programmers in class usage types.

One of the main limitations that can be pointed out to the technical discussion in this paper is that it does not cover a full formal treatment of the MOOL language, nor does it present proofs. However, we are confident that, with minor adjustments, we can adopt the techniques used in the work on modular session types [8] and apply them to our language.

A feature which our system does not consider is the treatment of exceptions. We do not predict any sort of mechanism for letting a method throw a checked exception, so that a client can execute some predefined error-handling code when it happens. However, Java-like error-management techniques, or alternatively, some exceptional default state encoded in the usage type (see [14]) could be easily added to our language.

Finally, although we tried to introduce some flexibility in our approach to linearity by letting linear objects evolve into shared ones, we still have not quite found a middle ground: our language entirely bans the aliasing of linear types, while completely allowing the aliasing of unrestricted types. Allowing limited forms of aliasing without loosing track of an object state is a topic that has occupied many researchers, and which we also plan to pursue in future work.

Acknowledgements This work was funded by project "Assertion-Types for Object-Oriented Programming", FCT (PTDC/EIA-CCO/105359/2008).

References

[1] Kevin Bierhoff & Jonathan Aldrich (2008): *PLURAL: checking protocol compliance under aliasing*. In: *ICSE Companion '08*, ACM Press, pp. 971–972, doi:10.1145/1370175.1370213.

[2] Joana Campos (2010): *Linear and Shared Objects in Concurrent Programming*. Master's thesis, University of Lisbon.

[3] Robert DeLine & Manuel Fähndrich (2003): *The Fugue protocol checker: Is your software Baroque?* Technical Report MSR-TR-2004-07, Microsoft Research.

[4] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopolou (2006): *Session Types for Object-Oriented Languages*. In: *ECOOP'06*, *LNCS* 4067, Springer, pp. 328–352, doi:10.1007/11785477_20.

[5] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern & Sophia Drossopolou (2005): *A Distributed Object-Oriented Language with Session Types*. In: *TGC'05*, *LNCS* 3705, Springer, pp. 299–318, doi:10.1007/11580850_16.

[6] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus & Steven Levi (2006): *Language Support for Fast and Reliable Message-based Communication in Singularity OS*. In: *EuroSys*, ACM Press, pp. 177–190, doi:10.1145/1217935.1217953.

[7] Cormac Flanagan & Martín Abadi (1999): *Types for Safe Locking*. In: *ESOP'99*, *LNCS* 1576, Springer, pp. 91–108, doi:10.1007/3-540-49099-X_7.

[8] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert & Alexandre Z. Caldeira (2010): *Modular Session Types for Distributed Object-Oriented Programming*. In: *POPL'10*, ACM Press, pp. 299–312, doi:10.1145/1706299.1706335.

[9] Simon J. Gay & Malcolm J. Hole (2005): *Subtyping for Session Types in the Pi Calculus*. Acta Informatica 42(2/3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[10] Simon J. Gay, António Ravara & Vasco T. Vasconcelos (2003): *Session Types for Inter-Process Communication*. Technical Report TR-2003-133, Comp. Sci., Univ. Glasgow.

[11] K. Honda, N. Yoshida & M. Carbone (2008): *Multiparty Asynchronous Session Types*. In: *POPL'08*, ACM Press, pp. 273–284, doi:10.1145/1328438.1328472.

[12] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR'93*, *LNCS* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[13] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP'98*, *LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[14] Filipe Militão (2008): *Design and Implementation of a Behaviorally Typed Programming System for Web Services*. Master's thesis, New University of Lisbon.

[15] Dimitris Mostrous (2005): *Moose: a Minimal Object Oriented Language with Session Types*. Master's thesis, University of London.

[16] Matthias Neubauer & Peter Thiemann (2004): *An Implementation of Session Types*. In: *PADL'04*, *LNCS* 3057, Springer, pp. 56–70, doi:10.1007/978-3-540-24836-1_5.

[17] Oscar Nierstrasz (1995): *Regular types for active objects*. In: *Object-Oriented Software Composition*, Prentice Hall, pp. 99–121.

[18] Robert E. Strom & Shaula Yemini (1986): *Typestate: A programming language concept for enhancing software reliability*. *IEEE Transactions on Software Engeneering* 12(1), pp. 157–171.

[19] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-Based Language and its Typing System*. In: *Parle'94*, *LNCS* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[20] Antonio Vallecillo, Vasco T. Vasconcelos & António Ravara (2006): *Typing the Behavior of Objects and Components using Session Types*. *Fundamenta Informaticæ* 73(4), pp. 583–598, doi:10.1016/S1571-0661(05)80382-2.

[21] Vasco T. Vasconcelos (2009): *Session Types for Linear Multithreaded Functional Programming*. In: *PPDP'09*, ACM Press, pp. 1–6, doi:10.1145/1599410.1599411.

[22] Vasco T. Vasconcelos (2009): *SFM*, chapter Fundamentals of Session Types, pp. 158–186. *LNCS* 5569, Springer, doi:10.1007/978-3-642-01918-0_4.

[23] Vasco T. Vasconcelos, Simon J. Gay & António Ravara (2006): *Typechecking a Multithreaded Functional Language with Session Types*. *Theoretical Computer Science* 368(1–2), pp. 64–87, doi:10.1016/j.tcs.2006.06.028.

[24] Vasco T. Vasconcelos, Simon J. Gay, António Ravara, Nils Gesbert & Alexandre Z. Caldeira (2009): *Dynamic Interfaces*. FOOL.